17-函数式编程：不用函数式编程语言，怎么写函数式的程序？

你好，我是郑晔！

前面几讲，我们讲了结构化编程和面向对象编程，对于大多数程序员来说，这些内容还是比较熟悉的。接下来，我们要讨论的函数式编程，对一些人来说就要陌生一些。

你可能知道，Java和C++已经引入了Lambda，目的就是为了支持函数式编程。因为，函数式编程里有很多优秀的元素，比如，组合式编程、不变性等等，都是我们值得在日常设计中借鉴的。即便我们使用的是面向对象编程语言，也可以将这些函数式编程的做法运用到日常工作中，这已经成为大势所趋。

但是，很多人学习函数式编程，刚刚知道了概念，就碰上了函数式编程的起源，遇到许多数学概念，然后，就放弃了。为什么学习函数式编程这么困难呢？主要是因为它有一些不同的思维逻辑，同时人们也缺少一个更好的入门方式。

所以，在这一讲中，我打算站在一个更实用的角度，帮你做一个函数式编程的入门。等你有了基础之后，后面两讲，我们再来讨论函数式编程中优秀的设计理念。

好，我们开始吧！

不断增加的需求

我们从一个熟悉的场景出发。假设我们有一组学生，其类定义如下：

```
// 单个学生的定义
class Student {
  // 实体 ID
  private long id;
  // 学生姓名
  private String name;
  // 学号
  private long sno;
  // 年龄
  private long age;
}

// 一组学生的定义
class Students {
  private List<Student> students
}
```

如果我们需要按照姓名找出其中一个，代码可能会这么写：

```
Student findByName(final String name) {
  for (Student student : students) {
    if (name.equals(student.getName())) {
        return student;
    }
  }
```

```
  return null;
}
```

这时候，新需求来了，我们准备按照学号来找人，代码也许就会这么写：

```
Student findBySno(final long sno) {
  for (Student student : students) {
    if (sno == student.getSno()) {
        return student;
    }
  }

  return null;
}
```

又一个新需求来了，我们这次需要按照 ID 去找人，代码可以如法炮制：

```
Student findById(final long id) {
  for (Student student : students) {
    if (id == student.getId()) {
        return student;
    }
  }

  return null;
}
```

看完这三段代码，你发现问题了吗？这三段代码，除了查询的条件不一样，剩下的结构几乎一模一样，这就是一种重复。

那么，我们要怎么消除这个重复呢？我们可以引入查询条件这个概念，这里只需要返回一个真假值，我们可以这样定义：

```
interface Predicate<T> {
  boolean test(T t);
}
```

有了查询条件，我们可以改造一下查询方法，把条件作为参数传进去：

```
Student find(final Predicate<Student> predicate) {
  for (Student student : students) {
    if (predicate.test(student)) {
```

```
        return student;
    }
  }

  return null;
}
```

于是，按名字查找就会变成下面这个样子（其他两个类似，就不写了）。为了帮助你更好地理解，我没有采用Java 8的Lambda写法，而用了你最熟悉的对象：

```
Student findByName(final String name) {
  return find(new Predicate<Student>() {
    @Override
    public boolean test(final Student student) {
      return name.equals(student.getName();
    }
  });
}
```

这样是很好，但你会发现，每次有一个新的查询，你就要做一层这个封装。为了省去这层封装，我们可以把查询条件做成一个方法：

```
static Predicate<Student> byName(final String name) {
  return new Predicate<Student>() {
    @Override
    public boolean test(final Student student) {
      return name.equals(student.getName();
    }
  }
}
```

其他几个字段也可以做类似的封装，这样一来，要查询什么就由使用方自己决定了：

```
find(byName(name));
find(bySno(sno));
find(byId(id));
```

现在我们想用名字和学号同时查询，该怎么办呢？你是不是打算写一个byNameAndSno的方法呢？且慢，这样一来，岂不是每种组合你都要写一个？那还受得了吗。我们完全可以用已有的两个方法组合出一个新查询来，像这样：

```
find(and(byName(name), bySno(sno)));
```

这里面多出一个and方法，它要怎么实现呢？其实也不难，按照正常的and逻辑写一个就好，像下面这样：

```
static <T> Predicate<T> and(final Predicate<T>... predicates) {
    return new Predicate<T>() {
        @Override
        public boolean test(final T t) {
            for (Predicate<T> predicate : predicates) {
                if (!predicate.test(t)) {
                    return false;
                }
            }

            return true;
        }
    };
}
```

类似地，你还可以写出or和not的逻辑，这样，使用方能够使用的查询条件一下子就多了起来，他完全可以按照自己的需要任意组合。

这时候，又来了一个新需求，想找出所有指定年龄的人。写一个byAge现在已经很简单了。那找到所有人该怎么写呢？有了前面的基础也不难。

```
Student findAll(final Predicate<Student> predicate) {
    List<Student> foundStudents = new ArrayList<Student>();
    for (Student student : students) {
        if (predicate.test(student)) {
            foundStudents.add(student);
        }
    }

    return new Students(foundStudents);
}
```

如此一来，要做什么动作（查询一个、查询所有等）和用什么条件（名字、学号、ID 和年龄等）就成了两个维度，使用方可以按照自己的需要任意组合。

直到现在，我们所用的代码都是常规的Java代码，却产生了神奇的效应。这段代码的作者只提供了各种基本元素（动作和条件），而这段代码的用户通过组合这些基本的元素完成真正的需求。这种做法完全不同于常规的面向对象的做法，其背后的思想就源自函数式编程。在上面这个例子里面，让代码产生质变的地方就在于Predicate的引入，而它实际上就是一个函数。

这是一个简单的例子，但是我们可以发现，按照“消除重复”这样一个简单的编写代码逻辑，我们不断地调整代码，就是可以写出这种函数式风格的代码。在写代码这件事上，我们常常会有一种殊途同归的感觉。

现在，你已经对函数式编程应该有了一个初步的印象，接下来，我们看看函数式编程到底是什么。

函数式编程初步

函数式编程是一种编程范式，**它提供给我们的编程元素就是函数**。只不过，这个函数是来源于数学的函数，你可以回想一下，高中数学学到的那个f(x)。同我们习惯的函数相比，它要规避状态和副作用，换言之，同样的输入一定会给出同样的输出。

之所以说函数式编程的函数来自数学，因为它的起源是数学家Alonzo Church发明的Lambda演算（Lambda calculus，也写作λ-calculus）。所以，Lambda这个词在函数式编程中经常出现，你可以简单地把它理解成**匿名函数**。

我们这里不关心Lambda演算的数学逻辑，你只要知道，Lambda演算和图灵机是等价的，都是那个年代对“计算”这件事探索的结果。

我们现在接触的大多数程序设计语言都是从图灵机的模型出发的，但既然二者是等价的，就有人选择从Lambda演算出发。比如早期的函数式编程语言LISP，它在20 世纪50年代就诞生了，是最早期的几门程序设计语言之一。它的影响却是极其深远的，后来的函数式编程语言可以说都直接或间接受着它的影响。

虽然说函数式编程语言早早地就出现了，但函数式编程这个概念却是John Backus在其1977 年图灵奖获奖的演讲上提出来。有趣的是，John Backus 获奖的理由是他在Fortran语言上的贡献，而这门语言和函数式编程刚好是两个不同“计算”模型的极端。

了解了函数式编程产生的背景之后，我们就可以正式打开函数式编程的大门了。

函数式编程第一个需要了解的概念就是函数。在函数式编程中，函数是一等公民（first-class citizen）。一等公民是什么意思呢?

- 它可以按需创建；
- 它可以存储在数据结构中；
- 它可以当作实参传给另一个函数；
- 它可以当作另一个函数的返回值。

对象，是面向对象程序设计语言的一等公民，它就满足所有上面的这些条件。在函数式编程语言里，函数就是一等公民。函数式编程语言有很多，经典的有LISP、Haskell、Scheme等，后来也出现了一批与新平台结合紧密的函数式编程语言，比如：Clojure、F#、Scala等。

很多语言虽然不把自己归入函数式编程语言，但它们也提供了函数式编程的支持，比如支持了Lambda的，这类的语言像Ruby、JavaScript等。

如果你的语言没有这种一等公民的函数支持，完全可以用某种方式模拟出来。在前面的例子里，我们就用对象模拟出了一个函数，也就是Predicate。在旧版本的C++中，也可以用functor（函数对象）当作一等公民的函数。在这两个例子中，既然函数是用对象模拟出来的，自然就符合一等公民的定义，可以方便将其传来传去。

在开头，我提到过，随着函数式编程这几年蓬勃的发展，越来越多的“老”程序设计语言已经在新的版本中加入了对函数式编程的支持。所以，如果你用的是新版本，可以不必像我写得那么复杂。

比如，在Java里，Predicate本身就是JDK自带的，and方法也不用自己写，加上有Lambda语法简化代码的编写，代码可以写成下面这样，省去了构建一个匿名内部类的繁琐：

```
static Predicate<Student> byName(String name) {
    return student -> student.getName().equals(name);
}

find(byName(name).and(bySno(sno)));
```

如果按照对象的理解方式，Predicate是一个对象接口，但它可以接受一个Lambda为其赋值。有了前面的基础，你可以把它理解成一个简化版的匿名内部类。其实，这里面主要工作都在编译器上，它帮助我们做了类型推演（Type Inference）。

在Java里，可以表示一个函数的接口还有几个，比如，Function（一个参数一个返回值）、Supplier（没有参数只有返回值），以及一大堆形式稍有不同的变体。

这些“函数”的概念为我们提供了一些基础的构造块，从前面的例子，你可以看出，函数式编程一个有趣的地方就在于这些构造块可以组合起来，这一点和面向对象是类似的，都是由基础的构造块逐步组合出来的。

我们讲模型也好，面向对象也罢，对于这种用小组件逐步叠加构建世界的思路已经很熟悉了，在函数式编程里，我们又一次领略到同样的风采，而这一切的出发点，就是“函数”。

总结时刻

这一讲我们讨论了**函数式编程**这种编程范式，它给我们提供的编程元素是函数。只不过，这个函数不同于传统程序设计语言的函数，它的思想根源是数学中的**函数**。

函数是函数式编程的一等公民（first-class citizen）。一等公民指的是：

- 它可以按需创建；
- 它可以存储在数据结构中；
- 它可以当作实参传给另一个函数；
- 它可以当作另一个函数的返回值。

如果你使用的程序设计语言不支持函数是一等公民，可以用其他的方式模拟出来，比如，用对象模拟函数。随着函数式编程的兴起，越来越多的程序设计语言加入了自己的函数，比如：Java和C++增加了Lambda，可以在一定程度上支持函数式编程。

函数式编程就是把函数当做一个个的构造块，然后将这些函数组合起来，构造出一个新的构造块。这样有趣的事情就来了。下一讲，我们来看看这件有趣的事，看函数式编程中是怎么组合函数的。

如果今天的内容你只能记住一件事，那请记住：**函数式编程的要素是一等公民的函数，如果语言不支持，可以自己模拟。**

思考题

今天我们开始了函数式编程的讲解，我想请你谈谈函数式编程给你留下的最深刻印象，无论是哪门函数式编程语言也好，还是某个函数式编程的特性也罢。欢迎在留言区分享你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言：

- 骨汤鸡蛋面 2020-07-03 08:05:34
  基于函数切分逻辑跟基于对象切分逻辑有何异同嘛？ [2赞]

  作者回复2020-07-03 09:36:08
  下一讲会讲函数式编程的组合性，会提到这个问题，简言之，函数式切分的是动词，面向对象切分的是名词。

- Jxin 2020-07-03 10:36:49
  demo里面，感觉消除重复不怎么明显（虽然调用侧少了恶心的ifelse，但整体代码量反而变多了）。反而适配的味道很重。将所有equest判断适配成统一规格的Predicate 函数。调用侧基于Predicate 函数（统一规格的接口）做职责链链式调用。挺复合统一多个类的接口设计的一应用场景。

- 王国庆 2020-07-03 09:38:49
  C 语言的函数指针是不是也是一种函数式编程呢，感觉也符合函数是 “一等公民” 的所有定义

- 被雨水过滤的空气 2020-07-03 09:25:55
  函数式编程给我印象最深的就是“纯函数”的概念。想想看，如果程序是由一个个纯函数组成的，那么在实现每一个函数的时候，不需要关心外界的情况，并且还可以利用缓存提高性能。

- 阳仔 2020-07-03 08:50:03
  函数式编程范式里，函数是一等公民，有了函数式编程就可以封装复用的逻辑代码，还可以组合这些逻辑代码
  语言设计越往后它是不断会进化的，毕竟这个世界里唯一不变的就是变化

  作者回复2020-07-03 09:37:07
  这个总结是到位的。

- bigben 2020-07-03 08:39:47
  java有空lambda之后好，代码精炼了很多，但有些人表示看不懂了，增加了理解难度

  作者回复2020-07-03 09:36:48
  用好 lambda 的第一条，不要在 lambda 里写太多代码。

- favorlm 2020-07-03 07:47:27
  在实际开发中，我一般把查询条件放到sql里，现在我准备拿到代码里，但是以前的分页插件阻止了我这么做，分页插件是以sql为基础的。那么请问郑大，如果改造查询用函数式表示的话，如何评判利弊呢。最近在拯救一个很烂(完全没有测试)的项目。